Chapter 1

VIRTUAL SUPERCOMPUTER AS BASIS OF SCIENTIFIC COMPUTING

Alexander Bogdanov[1]* *Alexander Degtyarev*[1] *Vladimir Gaiduchok*[2]
Ivan Gankevich[1] *Vladimir Korkhov*[1]

[1]Saint Petersburg State University
[2]Saint Petersburg Electrotechnical University «LETI»

Abstract

Nowadays supercomputer centers strive to provide their computational resources as services, however, present infrastructure is not particularly suited for such a use. First of all, there are standard application programming interfaces to launch computational jobs via command line or a web service, which work well for a program but turn out to be too complex for scientists: they want applications to be delivered to them from a remote server and prefer to interact with them via graphical interface. Second, there are certain applications which are dependent on older versions of operating systems and libraries and it is either non-practical to install those old systems on a cluster or there exists some conflict between these dependencies. Virtualization technologies can solve this problem, but they are not too popular in scientific computing due to overheads introduced by them. Finally, it is difficult to automatically estimate optimal resource pool size for a particular task, thus it often gets done manually by a user. If the large resource pool is requested for a minor task, the efficiency degrades. Moreover, cluster schedulers depend on estimated wall time to execute the jobs and since it cannot be reliably predicted by a human or a machine their efficiency suffers as well.

Applications delivery, efficient operating system virtualization and dynamic application resource pool size defining constitute the two problems of scientific computing: complex application interfaces and inefficient use of resources available — and virtual supercomputer is the way to solve them. The research shows that there are ways to make virtualization technologies efficient for scientific computing: the use of lightweight application containers and dynamic creation of these containers for a particular job are both fast and transparent for a user. There are universal ways to deliver application output to a front-end using execution of a job on a cluster and presenting

*E-mail address: {bogdanov,deg,vladimir}@csa.ru, {gajduchok,igankevich}@cc.spbu.ru

its results in a graphical form. Finally, an application framework can be developed to decompose parallel application into small independent parts with easily predictable execution time, to simplify scheduling via existing algorithms.

The aim of this chapter is to promote the key idea of a virtual supercomputer: to harness all available HPC resources and provide users with convenient access to them. Such a challenge can be effectively faced using contemporary virtualization technologies. They can materialize the long-term dream of having a supercomputer at your own desk.

Key words: virtual supercomputer, virtualization, distributed systems, heterogeneous systems, job scheduling.

AMS Subject Classification: 68M14, 68M20.

Introduction

A cloud, to wide extent, is an API. It is a certain contract model meeting the principles of the cloud computing. Five principles can be singled out:

- scalability,
- load balancing within the scalability,
- high availability up to being disaster-proof,
- easy access to resources from almost every place of the world and from any device,
- and payment on demand on the rental basis, i. e. on the most favorable terms for the client.

This means that it is necessary to support common standards between the clouds, thus being able to create hybrid clouds, which combine private and public computational resources. What helps to create hybrid clouds? One should be interested in products that both support the standards and provide open source licenses [29]. Eucalyptus and OpenNebula seem to be quite suitable: their API is backward compatible with Amazon [1, 2].

Our researches need the absence of vendor locks and have to be based on the open standards. The components have to be convenient, inexpensive and simple. Resource suppliers have to be interchangeable. Therefore, the result of the work will be universal and test-open for the other teams of scientists (program transferability, repeated use of components etc.). This is the service-oriented approach, when the components interact with the help of an API and the clients get the service that meets the contract guarantees.

So, we propose to use a cloud approach based on open standards and utilizing several up-to-date technologies, that make it very effective for large scale problems. Our main principles are

1. Cloud is determined completely by its API. And it is obvious from the user point of view, but the same must also be true from the point of view of different clouds interaction.

2. Operational environment must be UNIX-like. One of the main problems of computational GRIDs is load balancing and it is very difficult task since user is cut off from the resources. Partly this issue is resolved by problem solution environments, but many standard UNIX tools must be introduced into the API to make it really work.

3. Cloud uses protocols compatible with popular public clouds. Public clouds are not very useful for complex problems and the reason is clear — the more difficult your problem is, the more robust tools you must use. The universal tools cannot be used for complex problems. That is why specialized private clouds must be built for complex problems, but if its resources are not enough, some additional resources can be added from public cloud.

4. Cloud processes the data on the base of distributed file systems. The main problem with public clouds for data processing originates from the fact that for each computer in the cloud its own file system is used. That prevents both processing large data sets and scaling out the problem solution. To overcome this obstacle the distributed file system should be used in a private cloud, the type of which is determined by the nature of problem to be solved. If we add here the three ways of providing data consistency (Brewer's theorem) we can see that there are a lot of possibilities to organize the data processing, but only a few of those are really in use.

5. Consolidation of data is achieved by distributed DB. There are three levels of consolidation — servers, data and resources. It is more or less clear how server consolidation is done. Consolidation of data is more difficult and consolidation of resources is a real challenge to the cloud provider. We assume that most natural way to do this is to use Federal DB tools. Up to now we managed to do this by utilizing IBM's DB2 tools, but we believe that possibilities of latest PostgreSQL release will make it possible to work out freeware tool for such purpose [3].

6. Load balancing is achieved by the use of virtual processors with controlled rate. New high-throughput processors make it possible to organize virtual processors with different speed of computation. This opens natural possibility of making distributed virtual computational system with architecture adapted to computational algorithm and instead of mapping the algorithm onto the computer architecture we will match the architecture with the computational code.

7. Processing of large data sets is done via shared virtual memory. Actually all the previous experience shows that the only way to comfortably process large data sets is to use SMP system. Now we can effectively use shared memory tools (OpenCL) in heterogeneous environment and so make virtual SMP. The same tool is used for parallelization. The possibilities of a single image operational environment are also very effective.

8. Cloud uses complex grid-like security mechanisms. One of the cloud problems is security issues [19] but we feel that proper combination of GRID security tools with Cloud access technologies is possible.

Often virtualization is employed for many components of computing system such as processor, memory, storage subsystem and network interface, however, virtualizing all of them at once is not required in high performance computing where the efficiency of resulting computing system is of utmost importance. First, most of the virtualization comes at a cost of slight performance decrease, which grows with the size of the application. Second, virtualization of some components (i.e. network interfaces, processors) complicates the system's architecture without simplifying problem solution. This led to the scarce use of virtualization technologies in high performance computing; contrary to this, we feel that there is at least one way of using it to simplify architecture of the system and decrease its maintenance effort.

This way consists of storing applications which run on cluster and their dependencies inside lightweight containers. Upon submitting a task, containers are mounted on each host provided by a resource manager and parallel applications are executed inside each of them. The benefit of this approach is that a separate operating system and optimized libraries can be installed for each application without the need to alter host computer configuration. Additionally, it is easy to maintain different versions of applications (due to licensing or compatibility problems) in different containers and to update them in contrast to common shared-folder-for-all-applications configuration where there is a lot of manual work. Finally, since container virtualization does not imply processor virtualization overheads, this approach can be made efficient in terms of application performance.

The ramification of using application containers is that load balancing cannot be done efficiently with container migration, so object migration should be used instead. The main problem of container migration is that copies have too much unneeded data. In contrast to this, object migration which can be implemented in software framework or library is more efficient as the programmer not the system decides what to copy. Such framework was implemented based on event-driven programming approach and system load balancing is done through it.

Load balance is maintained by adjusting distribution of computational tasks among available processors with respect to their performance, and inability to distribute them evenly stem to arise not from technical reasons only, but also from peculiarity of a problem being solved. On one hand, load imbalance can be caused by heterogeneity of the tasks and inability to estimate how much time it takes to execute one particular task compared to some other task. Such difficulties arise in fluid mechanics applications involving solution of a problem with boundary conditions when the formula used to calculate boundary layer differs from the formula used to calculate inner points and takes longer time to calculate; the same problem arises in concurrent algorithms of intelligent systems that have different asymptotic complexities but solve the same problem concurrently hoping to obtain result by the fastest algorithm. On the other hand, load imbalance can be caused by heterogeneity of the processors and their different performance when solving the same problem and it is relevant when tasks are executed on multiple computers in a network or on a single computer equipped with an accelerator. Therefore, load imbalance can be caused by heterogeneity of tasks and heterogeneity of processors and these peculiarities should be both taken into account to maintain load balance of a computer system.

Related work

One of the first approaches to construct clusters from virtual machines was proposed in [16] and partially realized in In-VIGO [26], VMPlants [24], and Virtual Clusters on the Fly [27] projects. In-VIGO focused on the end-to-end design of a Web service that could employ VMs as part of a cluster computing system, while VMPlants and Virtual Clusters on the Fly focused on rapid construction of virtual clusters. All three systems were particularly concerned with the issue of specifying and adapting to requirements and constraints imposed by the user.

Dynamic Virtual Clustering (DVC) [15] implemented the scheduling of VMs on existing physical cluster nodes within a campus setting. The motivation for this work was to improve the usage of disparate cluster computing systems located within a single entity.

The idea of an adaptive virtual cluster changing its size based on the workload was presented in [10] describing a cluster management software called COD (Cluster On Demand), which dynamically allocates servers from a common pool to multiple virtual clusters.

Grid architecture that allows to dynamically adapt the underlying hardware infrastructure to changing Virtual Organization (VO) demands is presented in [28]. The backend of the system is able to provide on-demand virtual worker nodes to existing clusters and integrate them in any Globus-based Grid.

The goal of current work is to investigate possibilities provided by modern cloud and virtualization technologies to enable personal supercomputing meaning creation of dedicated virtual clusters based on user's and application's requirements. Unlike many of other projects in this area, our intention is to use created clusters of VM as a single resource provided to a single parallel application but not generating sets of worker nodes provided to different applications separately.

Research works on the subject of virtual clusters can be divided into two broad groups: works dealing with provisioning and deploying virtual clusters in high performance environment or GRID and works dealing with overheads of virtualization. Works from the first group typically assume that virtualization overheads are low and acceptable in high performance computing and works from the second group in general assume that virtualization has some benefits for high performance computing, however, the authors are not aware of the work that touches both subjects in aggregate.

In [11] authors evaluate overheads of the system for on-line virtual cluster provisioning (based on QEMU/KVM) and different resource mapping strategies used in this system and show that the main source of deploying overhead is network transfer of virtual machine images. To reduce it they use different caching techniques to reuse already transferred images as well as multicast file transfer to increase network throughput. Simultaneous use of caching and multicasting is concluded to be an efficient way to reduce overhead of virtual machine provisioning.

In [34] authors evaluate general overheads of Xen para-virtualization compared to fully virtualized and physical machines using HPCC benchmarking suite. They conclude that an acceptable level of overheads can be achieved only with para-virtualization due to its efficient inter domain communication (bypassing dom0 kernel) and absence of high L2 cache miss rate when running MPI programs which is common to fully virtualized guest machines.

In contrast to these two works the main principles of our approach can be summarized as follows. Do not use full or para-virtualization of the whole machine but use virtualization of selected components so that overheads occur only when they are unavoidable (i.e. do not virtualize processor). Do not transfer opaque file system images but mount standard file systems over the network so that only minimal transfer overhead can occur. Finally, amend standard task schedulers to work with virtual clusters so that no programming is needed to distribute the load efficiently. These principles are paramount to make virtualization lightweight and fast.

Event-driven architecture have been used extensively to create desktop applications with graphical user interface since MVC paradigm [23] was developed and nowadays it is also used to compose enterprise application components into a unified system with message queues [20, 33], however, it is rarely implemented in scientific applications. One example of such usage is GotoBLAS2 library [17, 18]. Although, it is not clear from the referenced papers, analysis of its source code[1] shows that this library uses specialized server object to schedule matrix and vector operations' kernels and to compute them in parallel. The total number of CPUs is defined at compile time and they are assumed to be homogeneous. There is a notion of a queue implemented as a linked list of objects where each object specifies a routine to be executed and data to be processed and also a number of CPUs to execute it on. Server processes these objects in parallel and each kernel can be executed in synchronous (blocking) and asynchronous (non-blocking) mode. So, compared to event-driven system GotoBLAS2 server uses static task scheduling, its tasks are not differentiated into production and reduction tasks, both the tasks and the underlying system are assumed to be homogeneous. GotoBLAS2 library exhibits competitive performance compared to other BLAS implementations [17, 18] and it is a good example of viability of event-driven approach in scientific applications. Considering this, our event-driven system can be seen as a generalization of this approach to a broader set of applications.

1. Virtual Supercomputer

1.1. Basic concepts

Virtual supercomputer can be seen as a collection of virtual machines working together to solve a computational problem much like a team of people working together on a single task. There is a known definition of personal supercomputer as a kind of metacomputer which was given in [30], however, in our approach virtual supercomputer is not only a personal supercomputer but it also offers a way of creating virtual clusters that are adapted to problem being solved and to manage processes running on these clusters (Figure 1). This is the case where virtual shared memory cannot be used directly because of high latency and low performance caused by complex data transfer patterns. Migration of processes to data as well as methods of workload balancing [22] can solve this problem and form a basis of load balancing technique for a virtual supercomputer.

Computers like people need some sort of collective board to share results of their work and advance problem solution one step further. In a distributed computing environment

[1] Source code is available in `https://www.tacc.utexas.edu/tacc-projects/gotoblas2/`.

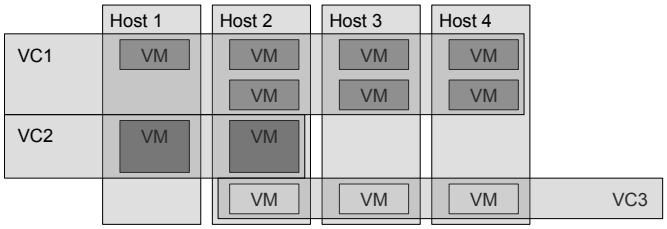

Figure 1. A cloud platform example with three virtual clusters over two physical clusters.

distributed file systems and distributed databases act as such a board, storing intermediate and final results of computation. Apart from a shared desk people in a team need some sort of management to solve a problem in time and computers need a way of combining them into hierarchy helping efficiently distribute tasks among available computing nodes. Finally, from a technical point of view, problem solution should be decoupled from actual execution of tasks by a virtualization layer as not every problem has efficient mapping on physical architecture of a distributed system. So, virtual supercomputer is not only a cluster of machines but also virtualization and middleware layers on top of it.

Although virtual supercomputer can be implemented in many ways and using different combinations of technologies, there are some principles that such implementation is considered to obey. On one hand these principles arise from similarity of different technologies and their implementations, on the other hand the purpose of some principles is to solve problems inherent to existing general-purpose distributed systems. In any case, the principles are useful for solving large-scale problems on virtual supercomputer and some of them can be neglected for problems of small sizes. Explanation of this principles follow.

Virtual supercomputer is completely determined by its application programming interface (API) and this API should be platform-independent. The use of API as the only interface in distributed processing systems is common, but its dependency on operating system or programming language leads to problems in the long run. For example, the first API for portable batch systems (PBS) was implemented in low-level C language and only for UNIX-like platforms which led to inability or inefficiency of its usage in other programming languages and in exposing it as a web service. Moreover, the API does not cover all the functions of underlying PBS [32]. So, using platform-independent API is one of the ways to avoid such integration and connectivity problems. In other words, API is a programming language of a virtual supercomputer and the only way of interacting with it.

Virtual supercomputer API provides functions to connect with other virtual supercomputers and such interaction is seamless. Interaction of different distributed systems is the way of solving large-scale problems [31] and seamless interaction helps compose hybrid distributed systems dynamically: to extend capacity when needed [8]. So it is the way of scaling virtual supercomputer to solve problems that are too complex for one virtual supercomputer.

Virtual supercomputer processes data stored in a single distributed database and this processing is done using virtual shared memory. Efficient data processing is achieved by distributing data among available nodes and by running small programs (queries) on each

host where corresponding data resides; this approach helps not only run query concurrently on each host but also minimizes data transfers [12, 25]. However, in existing implementations these programs are not general-purpose: they are parts of algorithm and they are specific to data model this algorithm was developed for. For example, in MapReduce framework programs represent map and reduce functions that are run on each row of table (or line of file) and it is difficult to compose general-purpose program to process any data within this framework [12]. On the other hand, virtual shared memory interface allows processing of data located on any host [6] and does it in efficient way. So, distributed database is a way of storing large data sets and virtual shared memory is a way of writing general-purpose program to process it.

Experiments show that using lightweight virtualization technologies (para-virtualization and application containers) instead of full virtualization is advantageous in terms of performance [7], hence virtual computing nodes should be created using lightweight virtualization technologies only. However, not every operating system supports these technologies and it should be possible to access virtual supercomputer facilities through fully-virtualized hosts. So, lightweight virtualization is inevitable in achieving balance between good performance and ease of system administration in distributed environment and as a consequence operating system should be UNIX-like for it to work.

Load balance is achieved using virtual processors with controlled clock rate and process migration. The first technique allows balancing coarse-granularity tasks and the second is suitable for fine-grained parallelism.

To summarize, virtual supercomputer is an API offering functions to run programs, to work with data stored in a distributed database and to work with virtual shared memory and this API is the only programming language of a virtual supercomputer.

1.2. Virtual workspace and small-scale virtual clusters

Virtual machine is the main building block of a virtual workspace. In its simplest form a workspace consists of a single virtual machine connected to storage and licensed software repository. If desired, resource capacity can be extended naturally by replicating virtual machine to form a virtual cluster. Cluster can be owned exclusively by a single user or shared by members of a whole research group. Moreover, considering large scale problem one can acquire resources of dedicated high performance machine (SMP or hybrid) or conventional cluster. Resource capacity extension occurs dynamically and acquired resources can be accessed from within single virtual machine.

Intended virtual machine usage is summarized as follows:

- solve scientific problems that fit into single virtual machine resources;
- access to computational resources, both clusters and dedicated machines;
- store experiment's data;
- develop applications (programming using commercial and open source compilers);
- perform other routine tasks.

Total processes	T-Platform cluster	SMP machine	Virtual cluster
32	84	42	35
64	66	36	48

Table 1. Crystal09 SrTiO3 test case wall clock time in minutes showing virtual cluster performance degradation. Virtual cluster characteristics: 16 virtual machines on 4×BL460c G7, 2×Intel X5675 CPUs and 96 GB RAM on one node, 64 cores total.

Private virtual cluster approach proved to be beneficial when using interactive resource-hungry software like Materials Studio or ADF. In that case computational resources of a single host exposed as a virtual machine are not enough for application to run smoothly, and virtual cluster boosts its performance. However, virtual cluster proved to be inefficient for solving large-scale problems with Crystal09. Running this application on virtual cluster puts heavy burden on network throughput. Careful investigation revealed that virtual cluster interconnect performance is degraded when multiple virtual nodes reside on a single physical host (Table 1). So private virtual cluster can be recommended for interactive small-scale applications used on the daily basis.

To summarize, para- and fully-virtual clusters are efficient at solving small-scale problems and offer rudimentary benefits of virtualization: ease of administration, improved security and resilience to hardware failures. The other use case is private virtual workspace.

1.3. Virtual clusters for high performance computing

Only lightweight virtualization technologies can be used to build virtual clusters for large-scale problems. This stems from the fact that on large scale no service overhead is acceptable if it scales with the number of nodes. In case of virtual clusters scalable overhead comes from processor virtualization which means that no para- and fully-virtualized machines are not suitable for large virtual clusters. This leaves only application container technologies for investigation. The other challenge is to make dynamic creation and deletion of virtual clusters take constant time.

Test system comprises many standard components which are common in high performance computing, these are distributed parallel file system which stores home directories with experiment's input and output data, cluster resource scheduler which allocates resources for jobs and client programs to pre- and post-process data; the non-standard component is network-attached storage exporting container's root files systems as directories. Linux Container technology (LXC) is used to provide containerisation, GlusterFS is used to provide parallel file system and TORQUE to provide task scheduling. The most recent CentOS Linux 7 is chosen to provide stable version of LXC (>1.0) and version of kernel which supports all containers' features. Due to limited number of nodes each of them is chosen to be both compute and storage node and every file in parallel file system is stored on exactly two nodes. Detailed hardware characteristics and software version numbers are listed in Table 2.

Creating virtual cluster in such environment requires the following steps. First, a client submits a task requesting particular number of cores. Then according to distribution of

Component	Details
CPU model	Intel Xeon E5440
CPU clock rate (GHz)	2.83
No. of cores per CPU	4
No. of CPUs per node	2
RAM size (GB)	4
Disk model	ST3250310NS
Disk speed (rpm)	7200
No. of nodes	12
Interconnect speed (Gbps)	1

Component	Details
Operating system	CentOS 7
Kernel version	3.10
LXC version	1.0.5
GlusterFS version	3.5.1
TORQUE version	5.0.0
OpenMPI version	1.6.4
IMB version	4.0
OpenFOAM version	2.3.0

Table 2. Hardware and software components of the system.

these cores among compute nodes a container is started on each node from the list with SSH daemon as the only program running inside it. Here there are two options: either start containers with network virtualization (using *macvlan* or *veth* LXC network type) and generate sufficient number of IP addresses for the cluster or use host network name space (*none* LXC network type) and generate only the port number to run ssh daemon on. The next step is to copy (possibly amended) node file from host into the first container and launch submitted script inside it. When the script finishes its work SSH daemon in every container is killed and all containers are destroyed.

For this algorithm to work as intended client's home directory should be bind-mounted inside the container before launching the script. Additionally since some MPI programs require *scratch* directories on each node to work properly, container's root file system should be mounted in copy-on-write mode, so that all changes in files and all the new files are written to host's temporary directory and all unchanged data is read from read-only network-mounted file system; this can be accomplished via Union or similar file system and that way application containers are left untouched by tasks running on the cluster.

To summarize, only standard Linux tools are used to build the system: there are no opaque virtual machines images, no sophisticated full virtualization appliances and no heavy-weight cloud computing stacks in this configuration.

1.4. Lightweight virtual clusters evaluation

To test the resulting configuration OpenMPI and Intel MPI Benchmarks (IMB) were used to measure network throughput and OpenFOAM was used to measure overall performance on a real-world application.

The first experiment was to create virtual cluster, launch an empty (with */bin/true* as an executable file) MPI program in it and compare execution time to ordinary physical cluster. To set this experiment up in the container the same operating system and version of OpenMPI as in the host machine was installed. No network virtualization was used, each run was repeated several times and the average was displayed on the graph (Figure 2). The results show that a constant overhead of 1.5 second is added to every LXC run after the 8^{th} core: one second is attributed to the absence of cache inside container with SSH

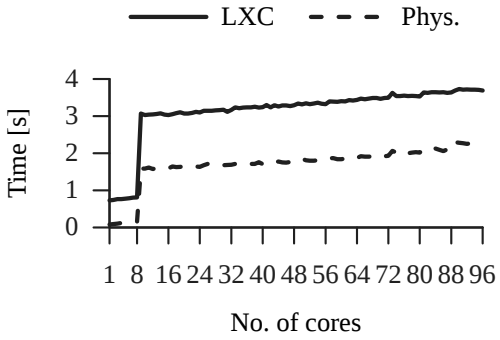

Figure 2. Comparison of LXC and physical cluster performance running empty MPI program.

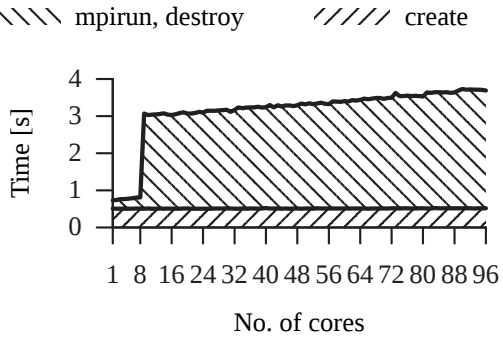

Figure 3. Breakdown of LXC empty MPI program run.

configuration files, key files and libraries in it and other half of the second is attributed to the creation of containers as shown in Figure 3. The jump after the 8^{th} core marks bounds of a single machine which means using network for communication rather than shared memory. The creation of containers is fully parallel task and takes approximately the same time to complete for different number of nodes. Overhead of destroying containers was found to be negligible and was combined with *mpirun* time. So, usage of Linux containers adds some constant overhead to the launching of parallel task depending on system's configuration which is split between creation of containers and filling the file cache.

The second experiment was to measure performance of different LXC network types using IMB suite and it was found that the choice of network virtualization greatly affects performance. As in the previous test container was set up with the same operating system and the same IMB executables as the host machine. Network throughput was measure with *exchange* benchmark and displayed on the graph (Figure 4). From the graph it is evident that until 214 bytes message size the performance is approximately the same for all network types, however, after this mark there is a dip in performance of virtual ethernet. It is difficult to judge where this overhead comes from: some studies report that under high load performance of bridged networking (*veth* is always connected to the bridge) is decreased [21], but IMB does not have high load on the system. Additionally, the experiment showed that as expected throughput decreases with the number of cores due to synchronization overheads (Figure 5).

The third and the last experiment dealt with real-world application performance and for this role the OpenFOAM was chosen as the complex parallel task involving large amount of network communication, disk I/O and high CPU load. The dam break RAS case was run with different number of cores (total number of cores is the square of number of cores per node) and different LXC network types and the average of multiple runs was displayed on the graph (Figure 6). Measurements for 4 and 9 cores were discarded because there is a considerable variation of execution time for these numbers on physical machines. From the graph it can be seen that low performance of virtual ethernet decreased final performance of OpenFOAM by approximately 5-10% whereas *macvlan* and *none* performance is close to the performance of physical cluster (Figure 7). So, the choice of network type is the

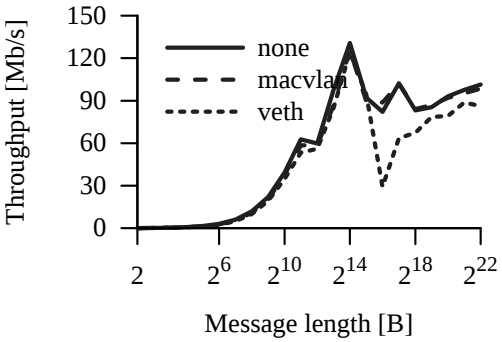

Figure 4. Average throughput of *exchange* MPI benchmark.

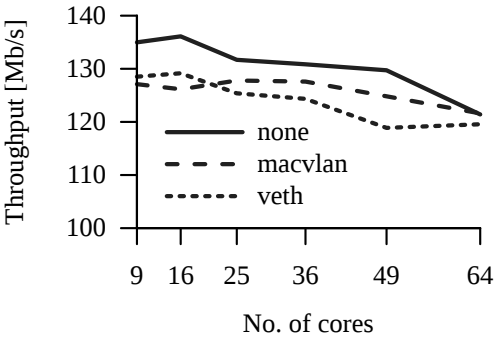

Figure 5. Throughput for 16Kb messages.

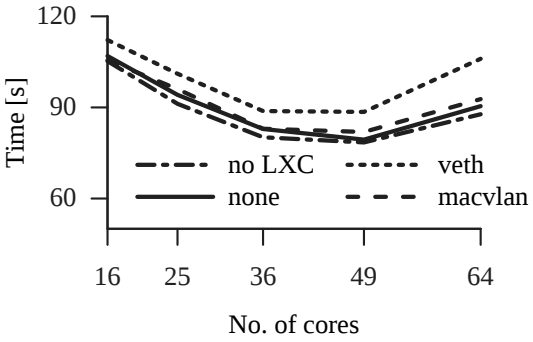

Figure 6. Average performance of OpenFOAM with different LXC network types.

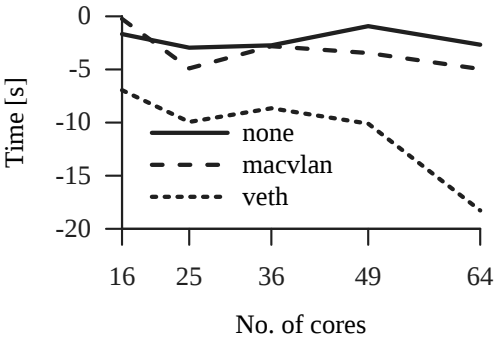

Figure 7. Difference of OpenFOAM performance on physical and virtual clusters. Negative numbers show slowdown of virtual cluster.

main factor affecting performance of parallel applications running on virtual clusters and its overhead can be eliminated by using *macvlan* network type or by not using network virtualization at all.

To summarize, there are two main types of overheads when using virtual cluster: creation overhead which is constant and small compared to average time of a typical parallel job and network overhead which can be eliminated by not using network virtualization at all.

Presented approach for creating virtual clusters from Linux containers was found to be efficient and its performance comparable to ordinary physical cluster: not only usage of containers does not incur processor virtualization overheads but also network virtualization overheads can be totally removed if host's network name space is used and network bandwidth saved by automatically transferring only those files that are needed through network-mounted file system rather than the whole images. From the point of view of system administrator storing each HPC application in its own container makes versioning and dependencies control easy manageable and their configuration does not interfere with the configuration of host machines and other containers.

2. Distributed scheduling

2.1. Load balancing

The most general and well-known approach to balancing the load on a multiprocessor system consists of decomposing data (or tasks) into homogeneous parts (or subtasks) and distributing them between available processor cores or computer cluster nodes, however, this approach is far from being the most efficient. Often, the overall number of parts is determined by the constraints of the problem being solved rather than computer architecture or cluster configuration being used. So, there are either too many parts compared to the number of processor cores resulting in process context switching and data copying overheads or there are too few parts to saturate all available processor cores. In addition to this, some problems can have non-homogeneous pattern of data decomposition which creates disbalance among the load on different processor cores. Finally, most of the computer systems in addition to processors consist of many disparate devices such as general purpose accelerators, GPUs and storage disks which play their roles in computation, so the final performance is limited by the performance of the slowest device. So, if the only thing which is taken into account when developing load balancing algorithm is its multi-core processor then the result is the first approximation to the ideal algorithm which can be improved by taking into account other devices of the system.

From mathematical point of view, load balance condition means equality of distribution function F of some task metric (e.g. execution time) to distribution function G of some processor metric (e.g. performance) and the problem of balancing the load is reduced to solving equation

$$F(x) = G(n), \tag{1}$$

where x is task metric (or time taken to execute the task) and n is processor metric (or relative performance of a processor needed to execute this task). Since in general case it is impossible to know in advance neither the time needed to execute the task on a particular processor, nor the performance of a processor executing a particular task, stochastic approach should be employed to estimate those values. Empirical distribution functions can be obtained from execution time samples recorded for each task: task metric is obtained dividing execution time by a number of tasks and processor metric is obtained dividing a number of tasks by their execution time. Also, any other suitable metrics can be used instead of the proposed ones, e.g. the size of data to be processed can be used as a task metric and processor metric can be represented by some fixed number.

It is easy to measure execution time of each task when the whole system acts as an event-driven system and an event is a single task consisting of program code to be executed and data to be processed. In this interpretation, load balancer component is connected to a processor recursively via profiler forming feedback control system. Profiler collects execution time samples and load balancer estimates empirical distribution functions and distributes new tasks among processors solving equation 1.

Static load balancing is also possible in this event-driven system and for that purpose a set of different load balancers can be composed into a hierarchy. In such hierarchy, distribution function is estimated incrementally from bottom to top and hierarchy is used to maintain static load balance. Physical processors are composed or decomposed into virtual

ones grouping a set of processors and assigning them to a single load balancer or assigning one physical processor to more than one load balancer at once. So static load balancing is orthogonal to dynamic load balancing and they can be used in conjunction.

To summarize, recursive load balancing approach targets problems exhibiting not only dynamic but also static imbalance and the balance can be achieved solving a single equation.

2.2. Hierarchical load balancing

To generalize the algorithm for multiple compute devices load balancing can be decomposed into two stages in order to cope with both heterogeneous problem decomposition pattern and saturation of heterogeneous devices. In the first step parts are distributed to devices according to their nature: computational tasks are assigned to either a processor or an accelerator depending on their implementation. Since storage devices do not have dedicated processors the core which is responsible for writing and reading data should be manually allocated for input/output tasks. The same core can be used for any device which is not capable of performing general purpose computations. In the second step when the device type is chosen the task is distributed to some device of this type by scheduling algorithm. Devices having the same type are almost always homogeneous so only the relative size of parts should be taken into account by the algorithm.

Backfill algorithm with certain modifications can be used as a scheduling algorithm for multi-processor system. This algorithm assumes that task execution time is known beforehand which is not the case for cluster schedulers: this time often manually estimated by people submitting the job to a queue [35]. Since there are many small tasks into which the problem is decomposed it is non-practical to estimate execution time of each task. Instead, task execution time can be reliably predicted by simple statistical means when the task is small. When a program is executed all tasks are assumed to be of the same size. Gradually, when timing is available for some significant number of tasks an average (or linear regression) of all times is computed and taken as prediction for next tasks.

Heterogeneity of devices can be taken into account using the same statistical approach, however, this time prediction should be based on the number of tasks executed by each device during some fixed unit of time.

Statistical approach is valid when the number of tasks is large and tasks are small which is not the case for cluster scheduling systems where tasks can execute for days or even weeks. Even for such systems Backfill algorithm works more efficient for small tasks [35]. There is some difficulty in defining which task is small and which is not so the best choice is to use natural problem decomposition. For example, fluid mechanics problems are often solved on some closed volume, so their natural decomposition is to create a task for each point of volume and then distribute them to processor cores. That way the larger the volume is the more saturated processor cores become and the more balanced the resulting load is. For small problems the load balance is not of a concern.

So, load balancing for multi-processor system is carried in two steps: first the task is sent to a device of appropriate type and then using modified Backfill scheduling algorithm is distributed to some of the devices of this type. Task execution time predictions are based on simple and fast statistical methods.

2.3. Event-driven system

The whole system was implemented as a collection of C++ classes, and problem-solving classes were separated from utility classes with an event-driven approach. In this approach, problem solution is represented by a set of executable objects or 'employees', each implementing a solution of one particular part of a problem. Each executable object can implement two methods. With the first method employee either solves part of the problem or produces child executable objects (or 'hires' additional employees) to delegate problem solution to them. Since upon completion of this method no object is destroyed, it is called 'production' task or 'upstream' task as it often delegates problem solution to a hierarchy layer located farther from the root than the current layer is. The second method collects execution results from subordinate executable objects and takes such object as an argument. Upon completion of this method the child object is destroyed or 'fired' so that the total number of executable objects is reduced. Hence this task is called 'reduction' task or 'downstream' task since the results are sent to a hierarchy layer located closer to root. Executable objects can send results not only to their parents but also to any number of other executable objects, however, when communication with a parent occurs the child object is destroyed and when the root object tries to send results to its nonexistent parent, the program ends. Execution of a particular object is performed via submitting it to a queue corresponding to a particular processor. Child and parent objects are determined implicitly during submission so that no manual specification is needed. Finally, these objects are never copied and are accessed only via their addresses. In other words, the only thing that is required when constructing an executable object is to implement a specific method to solve a task and object's life time is implicitly controlled by the system and a programmer does not have to manage it manually.

Execution of objects is carried out concurrently and construction of an executable object is separated from its execution with a thread-safe queue. Every message in a queue is an executable object and carries the data and the code needed to process it and since executable objects are completely independent of each other they can be executed in any order. There are real server objects corresponding to each queue in a system which continuously retrieve objects from a queue and execute their production or reduction tasks in a thread associated with the server object. Production tasks can be submitted to any queue, but a queue into which reduction tasks can be submitted is determined by a corresponding parent object so that no race condition can occur. Since each processor works with its own queue only and in its own thread, processing of queues is carried out concurrently. Also, each queue in a system represents a pipeline through which the data flows, however, execution order is completely determined by the objects themselves. So, executable objects and their methods model control flow while queues model data flow and the flows are separated from each other.

Heterogeneity of executable objects can cause load imbalance among different queues and this problem can be solved introducing imaginary (i.e. proxy) servers and profilers to aid in distribution of executable objects. Imaginary server is a server tied to a set of other servers and its only purpose is to choose the right child server to execute an object at.

In the simplest case, a proper distribution can be achieved with round-robin algorithm, i.e. when each arriving object is executed on the next server, however, in general case, some

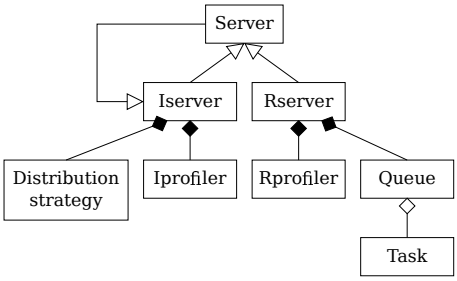

Figure 8. Class diagram for an event-driven system. 'Iserver' denotes imaginary server and 'Rserver' denotes real server.

additional information about completed runs is needed to choose the right server and this information can be collected with pluggable profiler objects. When a new object arrives to an imaginary server, actual profiling information is collected from child servers and specified distribution strategy is used to delegate execution of an object to an appropriate server, and some static distribution strategy is also possible. So, imaginary servers together with distribution strategies and profilers can be used to distribute executable objects among real servers taking into account some profiling information of completed object executions.

The class diagram of the whole event-driven system is depicted in the Figure 8 and the system works as follows.

1. When a program execution starts, the hierarchy of imaginary and real servers is composed. All real servers are launched in a separate threads and processing of executable objects starts.

2. The first object is created and submitted to the imaginary server at the top of the hierarchy. The server employs specified distribution strategy to choose an appropriate server from the next layer of the hierarchy to send the object to. The profiler gathers measurements of completed runs from subordinate servers and decides where to send an object.

3. The previous step repeats until the bottom level of the hierarchy is reached and real server which was found with the distribution strategy starts execution of an object.

4. Object is executed and measurements are made by a profiler. If during execution more executable objects are created and submitted to the top imaginary server, the whole algorithm is repeated for each new object; if the root object submits reduction task then all servers in the hierarchy are shut down, and program execution ends.

To sum up, the whole system is composed of the two hierarchies: one hierarchy represents tasks and data and their dependencies employing executable objects, the other hierarchy represents processing system employing imaginary and real server objects. Mapping of the first hierarchy to the second is implicit and is implemented using message queues. Such composition allows easy configuration of dynamic and static load distribution strategies and allows programming with simple executable objects.

2.4. Scheduling algorithm

Recursive load balancing was implemented as a load distribution strategy, however, equation 1 was not solved directly. The first problem occurring when solving this equation directly was that task metric x cannot be computed before actually running the task so it was estimated to be an average metric of a number of previous runs. The second problem was that when the task metric is known, the result of direct solution of equation 1 is not an identifier of a processor to execute the task on but it is number n – relative performance of a processor needed to execute the task and the number n is not particularly useful when determining where to execute the task. Therefore, equation 1 was not solved directly but its main idea was realized in an algorithm similar to round-robin.

The resulting algorithm works as follows.

1. First, algorithm collects samples recorded by profilers of child servers as well as estimates task metric and processor metric using values from previous runs. At this stage, not only averaging but also any other suitable predicting technique can be used.

2. Then, probability of having a task with metric equal to computed task metric is determined by counting samples equal to computed task metric and dividing it by the total number of samples.

3. The cursor pointing at the processor to execute the next task on is incremented by a step equal to a product of computed probability and computed processor metric.

4. Then, by recursively subtracting metric of each processor from the cursor, the needed processor is found and the task is executed on it.

The resulting mathematical formula for each step can be written simply as

$$cursor = cursor + F(\bar{x})\bar{n},$$

where $\bar{x}$ is a task metric and $\bar{n}$ is a processor metric. In case of fully homogeneous system and all tasks having equal metric this algorithm is equivalent to round-robin: all processors have metric equal to 1 and probability is always 1 so that the cursor is always incremented by 1.

Although, the algorithm is simple, in practice it requires certain modifications and a robust profiler to work properly. Since algorithm balances reciprocal values of task metric t (execution time) and processor metric $1/t$ (processor throughput), even a slight oscillation of a task metric can affect the resulting distributions greatly. The solution to this problem is to smooth samples with a logarithm function and it can be done in a straightforward way, because the algorithm does not make assumptions about metrics' dimensions and treat them as numbers. The second problem is that the algorithm should be implemented with integer arithmetic only to minimize overhead of load balancing. This problem can be solved by omitting mantissa after logarithm is applied to a sample and in that case processor metric is equal to task metric but has an opposite sign. The last major problem is that the distribution of task metric may change abruptly during program execution, which renders samples collected by a profiler for previous runs useless. This problem is solved by detecting a sharp change in task execution time (more than three standard deviations) and when an outlier is

detected the profiler is reset to its initial state in which distribution is assumed to be uniform. As a result of applying logarithm to each sample the algorithm becomes unsuitable for relatively small tasks and for tasks taking too much time, and although such tasks are executed, the samples are not collected for them as they often represent just control flow tasks. To summarize, the modified algorithm is implemented using integer arithmetic only, is suitable for relatively complex tasks and adapts to a rapid change of a task metric distribution.

One problem of the algorithm that stands aside is that it becomes inefficient in the event of high number of tasks with high metric values. It happens because when task is assigned to a particular processor it is not executed directly but rather gets placed in a queue. If this queue is not empty the task can reside in it for such a long time that its assignment to a particular processor will not match actual distribution function. The solution is simple: these stale tasks can be easily detected by recording their arrival time and comparing it with the current time and when such tasks are encountered by a queue processor, they can be redistributed to match the current distribution function. However, an existence of stale tasks is also an evidence that the computer is not capable of solving a problem fast enough to cope with continuously generated tasks and it is an opportunity to communicate with some other computers to solve the problem together. From a technical point of view, delegation of tasks to other computers is possible because tasks are independent of each other and read/write (serialization) methods are easily implemented for each of them, however, the problem was not addressed herein, and only load redistribution within a single computer was implemented.

Described algorithm is suitable for distributing production tasks, but a different algorithm is needed to distribute reduction tasks. Indeed, when executable objects come in pairs consisting of the child and its parent, all children of the parent must be executed on the same server so that no race condition takes place, so it is not possible to distribute the task on an arbitrary server but a particular server must be chosen for all of the child tasks. One possible way of choosing a server is by applying a simple hashing function to parent's memory address. Some sophistication of this algorithm is possible, e.g. predicting memory allocation and de-allocation pattern to distribute reduction tasks uniformly among servers, however, considering that most of the reduction tasks in tested program were simple (the reason for this is discussed in Section 2.5.) the approach seemed to be non viable and was not implemented. So, simple hashing algorithm was used to distribute reduction tasks among servers.

To summarize, recursive load distribution algorithm by default works as round-robin algorithm and when a reasonable change of task execution time is detected it automatically distributes the load in accordance with task metric distribution. Also, if there is a change in processor performance it is taken into account by relating its performance to other processors of computing system. Finally, if a task stays too much time in a queue it is distributed once again to match current distribution function.

2.5. Evaluation for compute-intensive problem

Event-driven approach was tested on the example of hydrodynamics simulation program which solves a real-world problem [4, 5, 13, 14]. The problem consists of generating real ocean wavy surface and computing pressure under this surface to measure impact of the

external excitations on marine object. The program is well-balanced in terms of processor load and for the evaluation purpose it was implemented using an introduced event-driven approach and the resulting implementation was compared to an existing non event-driven approach in terms of performance and programming effort.

Event-driven architecture makes it easy to write logs which in turn can be used to make visualization of control flow in a program. Each server maintains its own log file and whenever some event occurs, it gets logged in this file accompanied by a time stamp and a server identifier. Having such files available, it gets pretty straight-forward to reconstruct a sequence of events occurring during program execution and to establish connections between these events (to dynamically draw graph of tasks as they are executed). Many of such graphs are used in this section to demonstrate results of experiments.

Generation of a wavy surface is implemented as a transformation of a white noise, autoregressive model is used to generate ocean waves and pressures are computed using analytical formula. The program consists of preprocessing phase, main computer-intensive phase and post-processing phase. The program begins with solving Yule-Walker equations to determine autoregressive coefficients and white noise variance. Then a white noise is generated and gets transformed into a wavy surface. Finally, the surface is trimmed and written to an output stream. Generation of a wavy surface is the most computer-intensive phase which consumes over 80% of program execution time (Figure 18) for moderate wavy surface sizes and this time does not scale with a surface size. So, the program spends most of the time in the main phase generating a wavy surface (this phase is marked with $[G_0, G_1]$ interval in the graphs). The hardware used in the experiments is listed in Table 4. The program was tested in a number of experiments and then compared to other parallel programming techniques.

The first experiment consisted of measuring stale cycles and discovering causes of their occurrence. Program source code was instrumented with profiling directives and every occurrence of stale cycles was written to the log file. Also the total stale time was measured. Obtained results showed that stale cycles prevail in preprocessing and at the end of main phase but are not present in other parts of the program (Figure 9). The reason for this deals with insufficient amount of tasks available to solve during these phases which in turn is caused by global synchronizations occurring multiple times in preprocessing phase and naturally at the end of a program. Stale cycles in the main phase are caused by computation performing faster than writing results to disk: in the program only one thread writes data and no parallel file system is used. Further experiments showed that stale cycles consume at most 20% of the total execution time for 4 core system (Table 3) and although during this time threads are waiting on a mutex so that this time can be consumed by other operating system processes, there is also an opportunity to speed up the program. Considering file output performance stale cycles can only be reduced with faster storage devices combined with slower processors or with parallel file systems combined with fast network devices and interconnects. In contrast, the main cause of stale cycles in preprocessing phase deals with global synchronization and to minimize its effect it should be replaced by incremental synchronization if possible.

The next experiment consisted of measuring different types of overheads including profiling, load balancing, queuing and other overheads so that real performance of event-driven system can be estimated. In this experiment, the same technique was used to obtain mea-

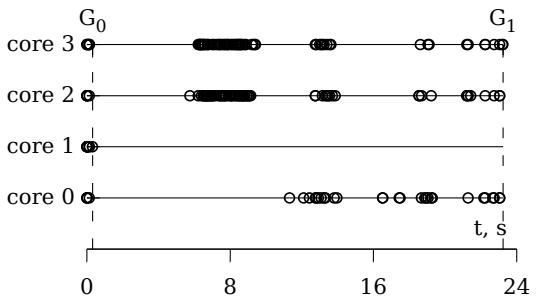

Figure 9. Occurrences of stale cycles in preprocessing and at the end of the main computational phase of a program. Range $[G_0, G_1]$ denotes computationally intensive phase.

Classifier	Time consumer	Time spent, %		
		4 cores	24 cores	48 cores
Problem solution	Production tasks	71	33	19
	Reduction tasks	13	4	2
Stale time	Stale cycles	16	63	79
Overhead	Load distribution overhead	0.01	0.0014	0.0017
	Queuing overhead	0.002	0.0007	0.0005
	Profiling overhead	0.0004	0.0004	0.0003
	Other overheads	0.06	0.03	0.02

Table 3. Distribution of wall clock time and its main consumers in event-driven system. Time is shown as a percentage of the total program execution time. Experiments for 4 cores were conducted on the system I and experiments for 24 and 48 cores were conducted on the system II from Table 4.

surements: every function causing overhead was instrumented and also the total time spent executing tasks and total program execution time was measured. As a result, the total overhead was estimated to be less than 0.1% for different number of cores (Table 3). Also the results showed that reduction time is smaller than the total time spent solving production tasks in all cases (Table 3). It is typical of generator programs to spend more time solving data generating production tasks than solving data processing reduction tasks; in a data-centric program specializing in data processing this relation can be different. Finally, it is evident from the results that the more cores are present in the system the more stale time is introduced into the program. This behavior was explained in the previous experiment and is caused by imbalance between processor performance and performance of a storage device for this particular computational problem. To summarize, the experiment showed that event-driven system and recursive load distribution strategy do not incur much overhead even on systems with large number of cores and the program is rather code-centric than data-centric spending most of its execution time solving production tasks.

In the third experiment, the total number of production tasks solved by the system was

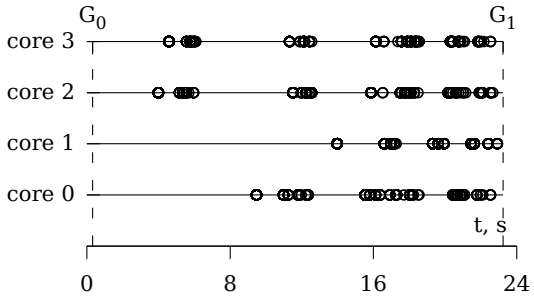

Figure 10. Event plot of resubmission of production tasks staying in a queue for too long time. Range $[G_0, G_1]$ denotes computationally intensive phase.

measured along with the total number of task resubmissions and it was found that there is high percentage of resubmissions. Each resubmission was recorded as a separate event and then a number of resubmissions for each task was calculated. The experiment showed that on average a total of 35% of tasks are resubmitted and analysis of an event log suggested that resubmissions occur mostly during the main computational phase (Figure 10). In other words 35% of production tasks stayed in a queue for too long time (more than an average time needed to solve a task) so underlying computer was not capable of solving tasks as fast as they are generated by the program. This result leads to a conclusion that if more than one computer is available to solve a problem, then there is a natural way to determine what part of this problem requires multiple computers to be solved. So, high percentage of resubmissions shows that machine solves production tasks slower than they are generated by the program so multiple machines can be used to speed up problem solution.

In the final experiment overall performance of event-driven approach was tested and it was found to be superior when solving problems producing large volumes of data. In the previous research it was found that OpenMP is the best performing technology for the wavy ocean surface generation [4], so the experiment consisted of comparing its performance to the performance of event-driven approach on a set of input data. A range of sizes of a wavy surface was the only parameter that was varied among subsequent program runs. As a result of the experiment, event-driven approach was found to have higher performance than OpenMP technology and the more the size of the problem is the bigger performance gap becomes (Figure 11). Also event plot in Figure 12 of the run with the largest problem size shows that high performance is achieved with overlapping of parallel computation of a wavy surface (interval $[G_0, G_1]$) and output of resulting wavy surface parts to the storage device (interval $[W_0, W_1]$). It can be seen that there is no such overlap in OpenMP implementation and output begins at point W_0 right after the generation of wavy surface ends at point G_1. In contrast, there is a significant overlap in event-driven implementation and in that case wavy surface generation and data output end almost simultaneously at points G_1 and W_1 respectively. So, approach with pipelined execution of parallelized computational steps achieves better performance than sequential execution of the same steps. In other words pipelined execution of sequential phases is an efficient way of exploiting nested (inter- and intra-device) parallelism of a problem.

Although OpenMP technology allows constructing pipelines, it is not easy to combine a pipeline with parallel execution of tasks. In fact such combination is possible if a thread-

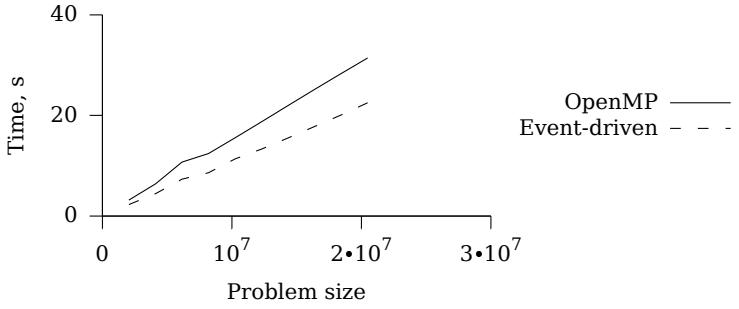

Figure 11. Performance comparison of OpenMP and event-driven implementations.

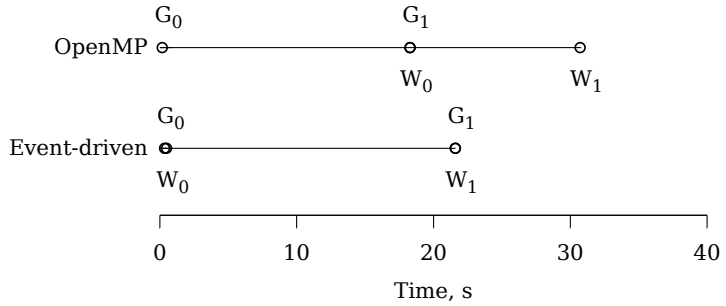

Figure 12. Event plot showing overlap of parallel computation $[G_0, G_1]$ and data output $[W_0, W_1]$ in event-driven implementation. There is no overlap in OpenMP implementation.

safe queue is implemented to communicate threads generating ocean surface to a thread writing data to disk. Then using *omp section* work of each thread can be implemented. However, implementation of parallel execution within *omp section* requires support for nesting *omp parallel* directives. So, combining pipeline with parallel execution is complicated in OpenMP implementation requiring the use a thread-safe queue which is not present in OpenMP standard.

To summarize, event-driven programming approach was applied to a real-world high-performance application and it was shown that it incurs low overhead, but results in appearance of stale periods when no problem solving is performed by some threads. The duration of these periods in the main phase can be reduced with faster storage equipment and the duration of stale periods in preprocessing phase can be reduced employing incremental synchronization techniques. Also, event-driven approach offers a natural way of determining whether program execution should scale to multiple machines or not, however, viability of such mode of execution was not tested in the present research. Finally, it was shown that event-driven approach is more efficient than standard OpenMP technology especially for large problem sizes and it was also shown that a pipeline combined with parallel execution works faster than sequential execution of parallelized steps.

Performance of recursive load distribution algorithm was compared to performance of round-robin algorithm and was tested in a number of scenarios with combinations of homogeneous and heterogeneous tasks and homogeneous and heterogeneous processors. In each experiment the total execution time and distributions of task metric and processor metric

Component	System	
Programming language	C++11	
Threading library	C++11 STL threads	
Atomics library	C++11 STL atomic	
Time measurement routines	`clock_gettime(CLOCK_MONOTONIC, ...)` `/usr/bin/time -f %e`	
Compiler	GCC 4.8.2	
Compiler flags	`-std=c++11 -O2 -march=native`	
	I	II
Operating system	Debian 3.2.51-1 x86_64	CentOS 6.5 x86_64
File system	ext4	ext4
Processor	Intel Core 2 Quad Q9650	2×Intel Xeon E5-2695 v2
Cores frequency (GHz)	3.00	2.40
Number of cores	4	24 (48 virtual cores)
RAM capacity (GB)	8	256
RAID device		Dell PERC H710 Mini
RAID configuration		RAID10
Storage device	Seagate ST3250318AS	4×Seagate ST300MM0006
Storage device speed (rpm)	7200	2×10000

Table 4. Testbed setup.

were measured and compared to uniform distribution case. All tests were performed on the same system (Table 4) and each scenario was run multiple times to ensure accurate results. Also, preliminary validation tests were performed to make sure that the algorithm works as intended. So, the purpose of evaluation was to demonstrate how algorithm works in practice and to measure its efficiency on a real problem.

It has already been shown that the algorithm consumes only a small fraction of total execution time of a program (Table 3), so the purpose of the validation test was to show algorithm's ability to switch between different task metric distributions. The switching is performed when a significant change (more than three standard deviations) of a task execution time occurs. The test have shown that the switching events are present in preprocessing phase and do not occur in the main phase (Figure 18). The cause of the switching is a highly variable task execution time inherent to preprocessing phase. So, profilers' resets occur only when a change of task execution time distribution is encountered and no switching is present when this distribution does not change.

The purpose of the first experiment was to show that the algorithm is capable of balancing homogeneous tasks on homogeneous computer and in that case it works like well-known round-robin algorithm. During the experiment, events of task submissions were recorded as well as additional profiler data and an event plot was created. In Figure 14 relative performance of each processor core is plotted and all the samples lie on a single line in

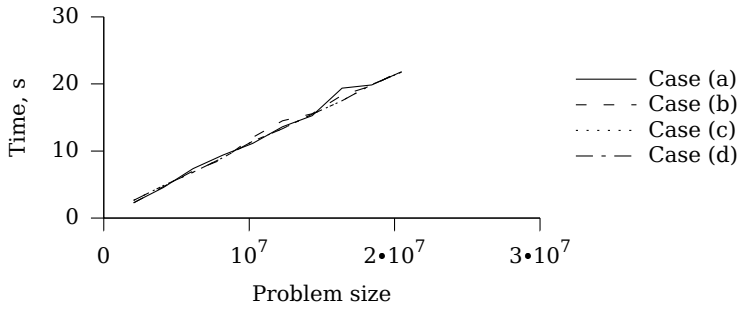

Figure 13. Performance comparison of different server configurations. Configurations are listed in Figure 18.

the computational phase. Since this phase consists of executing tasks of equal metric, the straight line represents the uniform distribution of tasks among processor cores constituting round-robin algorithm. So, in the simplest case of homogeneous tasks and processors recursive load balancing algorithm works as round-robin algorithm.

The purpose of the second experiment was to show that recursive load-balancing algorithm is capable of balancing homogeneous tasks on heterogeneous processors and in that case it can distribute the load taking into account performance of a particular processor. Although natural application of such load balancing is hybrid computer systems equipped with graphical or other accelerators, the experiment was conducted by emulating such systems with a hierarchy of servers. It was found that load balancing algorithm can recognize performance of different components and adapt distribution of tasks accordingly (Figure 15): I_1's first and second child servers have relative performance equal to 0.75 and 0.25 respectively whereas all children of I_2 server have relative performance equal to $\frac{1}{3}$. Also, this system setup shows performance similar to performance of the homogeneous computer configuration (Figure 13). So, recursive load balancing algorithm works on heterogeneous computer configurations and the performance is similar to homogeneous system case.

The purpose of the third experiment was to show that the algorithm is capable of balancing heterogeneous tasks on a homogeneous system and the experiment showed that performance gain is small. For the experiment the source code generating a wavy surface was modified so that parts of two different sizes are generated simultaneously. In order to balance such workload on a homogeneous system the step should be equal to $\frac{1}{2^i}n, i = 1,2,...$, where n is the processor metric (instead of being equal to 1 when parts have the same size) so that each processor takes two respective parts of the surface. In the Figure 16 showing results of the experiment the step reaches its optimal value of $\frac{1}{2}n$ (0.125 mark), however, it takes almost 8 seconds (or 40% of the total time) to reach this value. The first two cases do not exhibit such behavior and the step does not change during execution. Also, in the course of the experiment it was found that the step oscillates and to fix this it was smoothed with five point median filter and the number of samples was doubled. Finally, in subsequent experiments it was found that the more unique parts sizes are present in the main phase, the more samples should be collected to preserve the accuracy of the step evaluation, however, the increase in the number of samples led to slow convergence of the step to its optimal value. In other words, the more heterogeneous the tasks are, the more time is needed to find

the optimal step value for them.

The purpose of the fourth and final experiment was to show that the algorithm is capable of balancing heterogeneous tasks on heterogeneous system and results were similar to the previous experiment. System configuration was the same as in the second experiment. Although, in the Figure 17 showing the results metrics and steps of both servers reach nearly optimal values, there are more disturbances in these processes. So, the algorithm works with heterogeneous tasks and system but heterogeneity of a system increases variability of the step. In other words, heterogeneity of a system also increases time needed to find the optimal step value.

To summarize, from the experiments one can conclude, that the algorithm works on any system configuration and with any task combination, but requires tuning for a particular problem. However, experience obtained in the course of the experiments suggests that not only heterogeneity of tasks and computers increases the number of samples and convergence time but also there are certain task size distributions that cannot be handled efficiently by this algorithm and can extend this time indefinitely. One example of such distribution is linearly increasing task size. In this case step is always equal to $1/m$, where m is the number of samples, and there is no way to tune the algorithm to balance such workload. So, the downside of recursive load balancing algorithm is that it is suitable for closed metric distributions with low variability of the metric and more general and simple modified Backfill algorithm can be used in other cases. Also, it is evident from the experiments from the Section 2.5. that in the tested program the dominating performance factor is balance between the speed of wavy surface generation and the speed of writing it to storage device. In that case, load balancing algorithm plays only a second role and any combination of computer and task heterogeneity demonstrates comparable performance as was depicted in Figure 13.

2.6. Evaluation for mixed compute- and data-intensive problem

The problem of classification of wave energy spectra is both data- and compute-intensive which makes it on one hand amenable to data-centric programming approaches like Hadoop and on the other hand to parallel programming techniques. In the *mapping* phase spectra should be pre-processed and converted to some convenient format and in the *reduction* phase resulting spectra are classified using genetic optimisation algorithm. These steps represent general algorithm for data processing with Hadoop, however, classification algorithm is itself parallel which makes the problem of classification difficult to program in Java (the language in which Hadoop programs are usually written). Therefore, we feel that Hadoop is not the most efficient way to solve the problem and a distributed program which mimics useful Hadoop behaviour should be used instead.

The NDBC dataset[2] consists of spectra which are sorted by year and station where measurements were acquired. Data for each spectrum is stored in five variables which are used to reconstruct original frequency-directional spectrum with the following formula:

$$S(\omega,\theta) = \frac{1}{\pi}\left[\frac{1}{2} + r_1\cos(\theta-\alpha_1) + r_2\sin(2(\theta-\alpha_2))\right] S_0(\omega,\theta).$$

[2] http://www.ndbc.noaa.gov/dwa.shtml

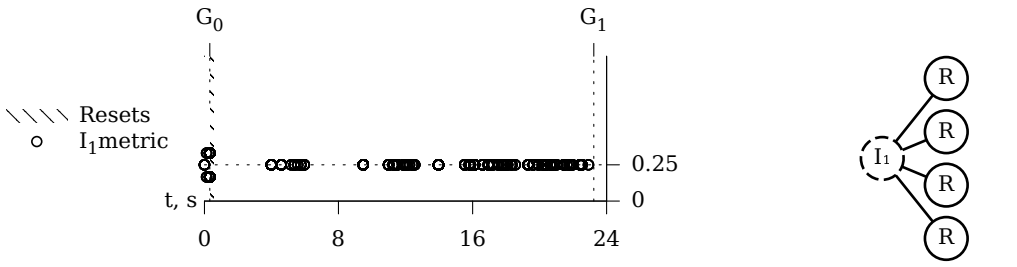

Figure 14. Homogeneous tasks and homogeneous computer case.

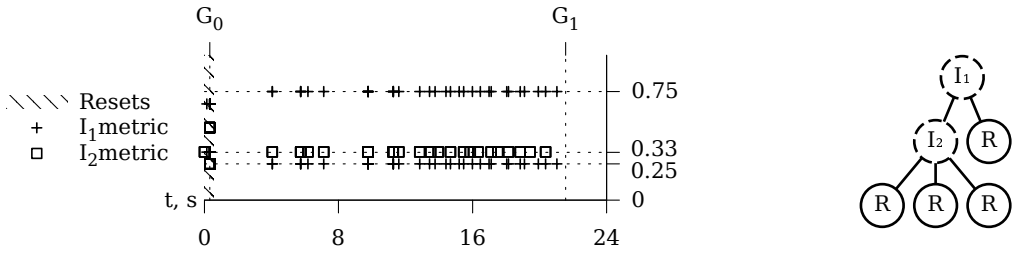

Figure 15. Homogeneous tasks and heterogeneous computer case.

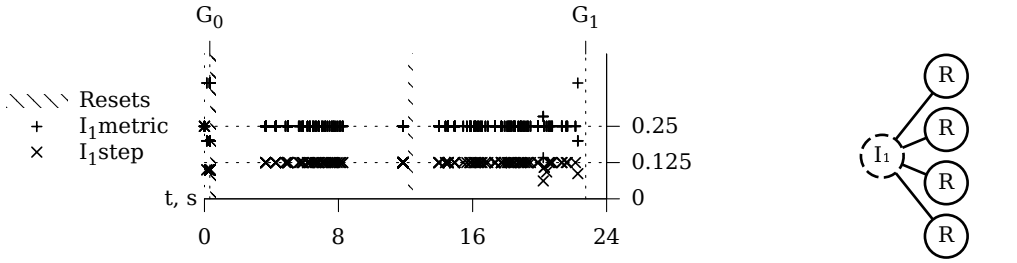

Figure 16. Heterogeneous tasks and homogeneous computer case.

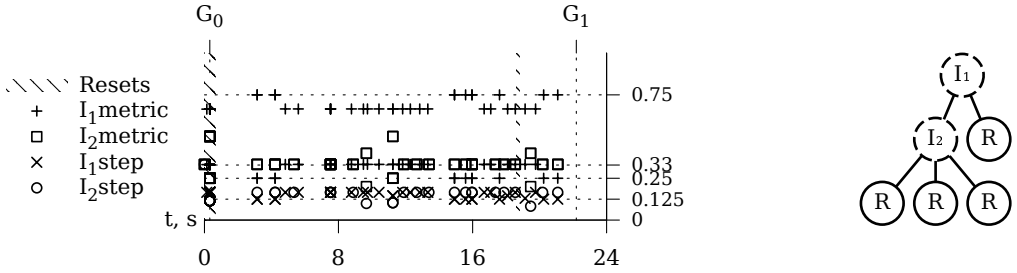

Figure 17. Heterogeneous tasks and heterogeneous computer case.

Figure 18. Event plot of task submissions and relative performance of child servers recorded at the time of submissions. *I* denotes 'Iserver' and *R* denotes 'Rserver'. Profiled servers are marked with dashed line.

Property	Details
Dataset size	144MB
Dataset size (uncompressed)	770MB
Number of wave stations	24
Time span	3 years (2010–2012)
Total number of spectra	445422

Table 5. Dataset properties.

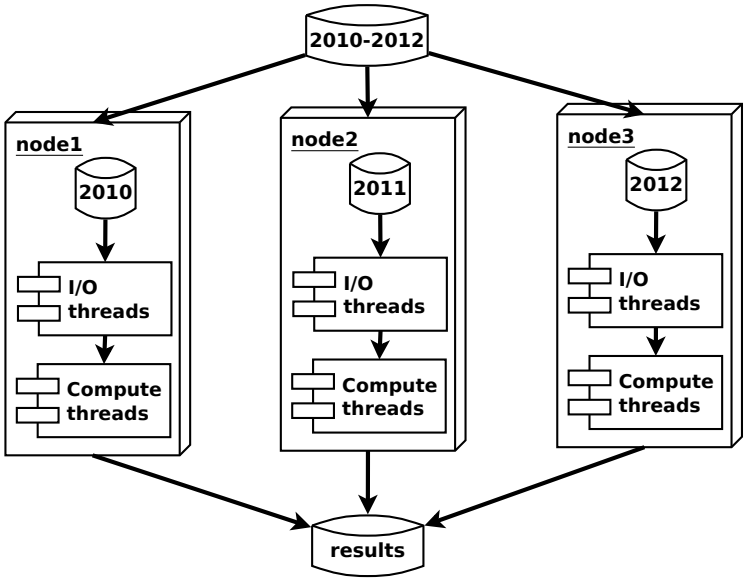

Figure 19. Implementation diagram for distributed pipeline.

Here ω denotes frequency, θ is the wave direction, $r_{1,2}$ and $\alpha_{1,2}$ are parameters of spectrum decomposition and S_0 is the measured wave energy spectrum. Detailed properties of the dataset used in evaluation are listed in Table 5.

The algorithm of processing spectra is as follows. First, current directory is recursively scanned for input files. All directories are recursively distributed to processing queues of each machine in the cluster. Processing begins with joining corresponding measurements for each spectrum variables into a single tuple which is subsequently classified by a genetic algorithm (this algorithm is not discussed in the paper and in fact can be replaced by any other suitable classification algorithm). While processing results are gradually copied back to the machine where application was executed and when the processing is complete the programme terminates. The resulting implementation is shown in Figure 19.

Directory structure can be arbitrary and the only thing it serves is to distribute the data, however, files containing corresponding measurements should be placed in a single directory so that no joining of variables residing in different machines can happen. In this test spectra were naturally sorted into directories by year and station.

The feature which makes this implementation different from other similar approaches is

Component	Details
CPU model	Intel Q9650
CPU clock rate (GHz)	3.0
No. of cores per CPU	4
No. of CPUs per node	1
RAM size (GB)	4
Disk model	ST3250318AS
Disk speed (rpm)	7200
No. of nodes	3
Interconnect speed (Mbps)	100

Component	Details
Operating system	Debian Linux 7.5
Hadoop version	2.3.0
GCC version	4.7
Compiler flags	`-std=c++11`

Table 6. Hardware and software components of the system.

that both processors and disks work in parallel throughout the programme execution. Such behaviour is achieved with assigning a separate thread (or thread pool) for each device and placing tasks in the queue for the corresponding device in this pool. As tasks that read from the disk complete they produce tasks for CPUs to process the data which was read and place them into a processor task queue. In similar way when tasks processing data complete they place tasks to write the data into disk task queue. In similar vein via a separate task queue network devices transmit the data to a remote node. So, each device has its own thread (or thread pool) and all of them work in parallel by placing tasks in each other's task queues. Since tasks «flow» from one queue to another and queues can reside on different machines this approach is called distributed pipeline.

The system setup which was used to test the implementation consisted of commodity hardware and open-source software (Table 6) and evaluation was divided into two stages. In the first stage Hadoop was installed on each node of the cluster and was configured to use host file system as a source of data so that performance of parallel file system which is used by default in Hadoop can be factored out from the comparison. To make this possible the whole dataset was replicated on each node and placed in the directory with the same name. In the second stage Hadoop was shut down and replaced by newly developed application and dataset directories were statically distributed to different nodes to nullify the impact of parallel file system on the performance.

In the test it was found that Hadoop implementation has low scalability and a performance of approx. 1000 spectra per second and alternative implementation has higher scalability and performance of approx. 7000 spectra per second (Figure 20). The source of such inefficiency was found to be temporary data files which are written to disk by Hadoop on each node. These files represent sorted parts of the key-value array and are part of implementation of merge sort algorithm used to distribute the keys to different nodes. For NDBC dataset the total size of these files exceeds the size of the whole dataset which appears to be the consequence of Hadoop not compressing intermediate data (the initial dataset has compression ratio of 1:5). So, the sorting algorithm and non careful work with compressed data lead to performance degradation and inefficiency of Hadoop for NDBC dataset.

The sorting is not needed to distribute the keys and in distributed pipeline directory hierarchy is used to determine machine for reduction. For each directory a separate task is

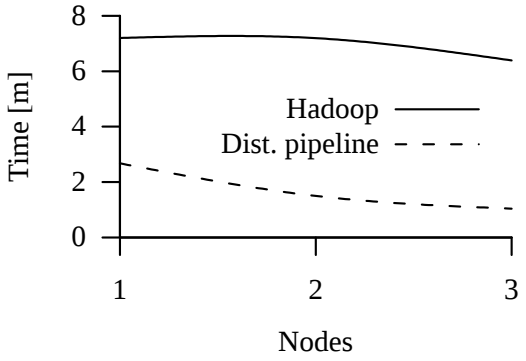

Figure 20. Performance of Hadoop and distributed pipeline implementations.

created which subsequently creates tasks for each sub-directory and each file. Since each task can interact with its parent when the reduction phase is reached reduction tasks are created on the machines where parents were executed previously.

3. Scheduling legacy applications

The approach discussed in the previous section offers a novel way of parallelization of programs and requires the use of new API. However, rewriting existing well-tested code which uses established standards (e.g. MPI, OpenMP) is not always viable. Developers are resistant to change and tend to use traditional tools even if they are obsolete and show relatively poor performance. As a result, transition to new standards takes time.

Although, there are some established standards in industry and scientific computing, there is no universal one. Each of them has its own application area such as big data processing or parallel and geographically distributed computations, but their convergence continues to make universal standard for distributed computing a reality. For now, standards usually have several implementations that sometimes vary in terms of performance and capabilities. That is why one should consider a multitude of heterogeneous applications that use different approaches and standards when planning scientific computational infrastructure.

The next difference between applications is a software distribution model. There is a good tradition in scientific community to develop and use open source software packages for its own needs. However, sometimes researchers can not find an open source alternative to a proprietary closed source program or such program has capabilities which can not be found in the open source one. A closed source application is a «black box»: developers often do not take time to properly describe algorithms that are implemented in the program or it can be a proprietary secret. For a scheduler it makes difficult to estimate efficiency of such an application.

To summarize, HPC resource in general can be shared by different groups of scientists that use open and closed source software packages, that in turn use various standards which makes efficient resource sharing a challenging problem. One should take into account every scientific application separately. Solving this problem seems to be impossible without in-

telligent software. Common schedulers and resource allocation systems are limited in their ability to estimate program execution times and their granularity is not fine enough to make sharing of resources possible.

Our approach is to profile application run, analyze the data with simple statistical methods and gradually improve future system utilization with intelligent resource reallocation. This approach makes possible to optimize performance of legacy applications source code of which can not be changed for some reason.

3.1. Resource allocation

Distributed resources can be managed in several ways.

- **No management system.** Administrator installs operating system and grants access to nodes to users. It is the simplest approach from infrastructure point of view, but it is really difficult to do resource planning in such system since inexperienced users would probably start their jobs on the first node.
- **Cloud.** Administrator installs a hypervisor on each cluster node and creates a cloud using some cloud platform (e.g. OpenStack). Although, cloud computing simplifies infrastructure management its overhead due to virtualization and virtual machine migration can not be underestimated for efficiency of high-performance applications.
- **Single system image.** Administrator installs MOSIX or ScaleMP on the entire cluster. Such systems are quite convenient for users but this does not come without pitfalls: even though virtual SMP is a simple abstraction the fact that resources are still remote at the physical layer makes it difficult to scale to really large number of nodes. For example, failure of a node in vSMP causes a whole virtual system restart to exclude its processors from the kernel.
- **Classical resource management system.** Administrator installs a portable batch system (TORQUE, PBS Professional) or similar systems (HTCondor or Univa Grid Engine). Despite their age the use of PBS is still common and it is the main reason that evolution of this type of systems forms a basis of a scheduler for legacy applications.

Portable batch systems became somewhat classical in the world of scientific computing. Many universities across the world use one of PBS implementations or similar systems to schedule parallel jobs on their clusters. Why these systems became so popular and widespread in the scientific community? They are simple. PBS implementations can reserve computational resources like CPU cores, memory, GPUs or disk space for job execution. One can view PBS as a generalization of Linux *at* command that uses a scheduler to decide the exact time and the list of nodes to execute a job. If there are not enough free resources then job is put into a queue.

Such systems usually provide administrators with scarce accounting information and granularity of jobs is not fine: each of them is a long-running parallel application. Common monitoring systems can collect more information but it is mapped to processes not jobs. That is why collecting profiling data from a more «verbose» scheduler can be useful to reallocate resources and minimize their usage for subsequent parallel application runs.

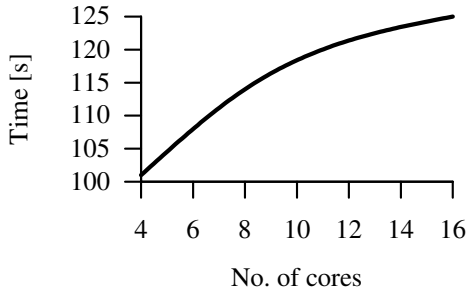

Figure 21. Limited scalability of an OpenFOAM 2.2 test case due to Amdahl's law.

	TCP/IP	IPoIB	RDMA
Network device	2x Ethernet 1G	Infiniband QDR 4x	Infiniband QDR 4x
NIC speed [Gb/s]	2	40	40
PingPong speed [Gbit/s]	1.2	8.9	25.3
MPI library	OpenMPI 1.6.4	OpenMPI 1.6.4	OpenMPI 1.6.4

Table 7. Performance of Intel MPI «PingPong» benchmark for TCP/IP, IPoIB and RDMA protocols.

For example, if user submits certain job to a PBS queue several times, performance may vary despite the fact that the same resources are requested each time. Job's execution time depends on mapping of processes to cluster nodes (all processes on one or several nodes), network characteristics in case of several nodes and real processor load. Although, CPU cores are reserved according to user requests, some shared resources like network can be impacted by the number of currently running jobs. There is a possibility that some jobs overload or underload the system, that is, use much more or much less resources than requested or use them inefficiently due to Amdahl's law (Figure 21).

Without knowledge of the underlying levels that a specific application use resource utilization can be inefficient. Table 7 illustrates test run of an MPI application: first time using MPI implementation that works via RDMA and second time corresponds to the another implementation that can work only with IPoIB (IP over Infiniband). Without complex monitoring system it is difficult to track such situations to amend MPI library configuration.

It is easy to understand PBS resource reservations, however, their coarse granularity does not reflect the actual load. For example, 64 cores can be requested for a program that uses only a few of them due to erroneous invocation or insufficient problem size. Additionally, there is no need to specify resources manually when optimal reservation is known beforehand and can be assigned by the system automatically. If a cluster have free resources they can be dynamically reallocated for a job. Continuous resource reallocation and finding optimal reservation is the core ideas of the new scheduler.

3.2. New scheduler for legacy applications

The new scheduler is intended for legacy applications, that is, applications source code of which can not be rewritten for some reason. These programs can be scheduled efficiently

Component	Details
CPU	Intel Xeon X5650
No. of nodes	24
No. of CPUs	2
No. of CPU cores	6
CPU freq. [GHz]	2.67
RAM [GB]	96
Network device	InfiniBand QDR 4x
	2x Ethernet 1Gb
Operating system	CentOS 6.4

Table 8. Test platform characteristics.

without recompilation. The main goal of this scheduler is to improve overall resource utilization and overcome common PBS drawbacks discussed in the previous section while maintaining low overhead of dynamic scheduling.

The main features of the system are listed below.

- **Flexible resource reservation and dynamic resource reallocation.** Instead of fixed resource reservation (the way PBS works) we can make an initial resource allocation for a job but constantly monitor the job and reallocate resources in case of underload or overload. In case of underload we can run another job on the same resources while in case of overload we have three options: keep the job running if there are free resources available and log this information for calculating user rating, suspend this job to run another one (and continue the first job when more resources become free) or completely stop the job and warn the user. This improves resource utilization when the system has some bottlenecks or is being used by inexperienced users.

- **Detailed accounting.** For each process in the job the statistics is gathered and analyzed to find possible bottlenecks and make predictions for optimal resource reservation. Applications can be profiled without necessity to go deeply into the source code and even closed source applications can be profiled.

- **Predictions module.** This module can collect a lot of statistical information from runs of different software packages which in the long term can be used to find optimal resource reservation and improve overall resource utilization. Performance often depends on the specific task so only general statistical information can be gathered. For example, if 10% fraction of a program is sequential, then allocating 128 CPU cores instead of 16 gives up to 1.5 speedup according to Amdahl's law. This fact can be figured out from statistical information to inform users upon submission of a similar job. Another example is two MPI implementations: one uses RDMA protocol and other uses IPoIB. If some application uses the second MPI implementation the good choice is to allocate nodes without Infiniband NICs for that program since IPoIB is slower than RDMA. Finally, from statistics applications performing many I/O operations can be found to redirect similar jobs to nodes with fast storage in the

future. All these can be used to give advises to users or automatically apply additional rules for user jobs (i.e. to adjust the amount of resources). That way user should not care about exact resources that are needed to execute the job and their exact amount. Jobs become «virtual» in a sense that they require no manual resource allocation, all the resource management is automatic and over time becomes optimal for each application.

- **User rating.** This simple feature is rarely used in scientific clusters, but is often used in public clusters and similar areas. Such rating shows how well a user can guess the optimal resource allocation and thus reflects overall efficiency of user jobs. This rating can be used to improve resource utilization: if a job consumes more resources than it was requested or the requested resource pool is much larger than needed, the rating of a user is decreased. There can be different coefficients for underload and overload. So, the rating reflects the overall user experience in running scientific software on clusters and its goal is to implement priority and limitation policies: rating defines job priority and limits resource usage for each user. The rating accelerates finding of optimal resource reservation with help of the most experienced users.

- **The possibility to use profiling modules with existing PBS installation.** Accounting module can be seamlessly used in existing PBS cluster. In this case, administrator should rewrite *prologue* to pass user script as an argument to profiling module.

- **Use of low-level native API.** There are two APIs that are used in the profiler: *ptrace* operating system call and Pthreads library. Jobs are monitored, suspended and started using *ptrace*. Accounting module uses fork: the main process forks a profiler that monitors child processes and collects data and then the main process calls *ptrace* with *PTRACE_TRACEME* option and executes a user program with *execve* function. The profiler is notified when the main process creates or terminates child processes to collect profiling information from proc virtual file system. This approach is native for Linux but not for UNIX distributions.

The work flow of such system can be described as follows (Figure 22). A job is started under the control of the profiling program. Profiling is triggered by a timer and information is gathered for all nodes. When a new job is submitted, the server program looks for suitable nodes for this job and invokes prediction module to optimize resource reservation. If there are no free resources the server checks resource utilization of running jobs. If it finds that some job uses fewer resources than requested the server can reallocate these resources for a new job. However, if the resource utilization is increased the new job will be suspended.

So, in addition to the classical queue and nodes with running jobs there are some suspended jobs that can be resumed immediately. The mechanism is akin to kernel process management at a scale of the whole cluster. The behavior can be adjusted by different policies and changed by administrator in real time (for example, unimportant jobs can be suspended manually to start some mission critical one).

Since the profiler is an intermediate program that forks a real one, there is no need to run background services on each node. On a remote node profiler is invoked by amending SSH command that MPI uses to launch remote parallel processes.

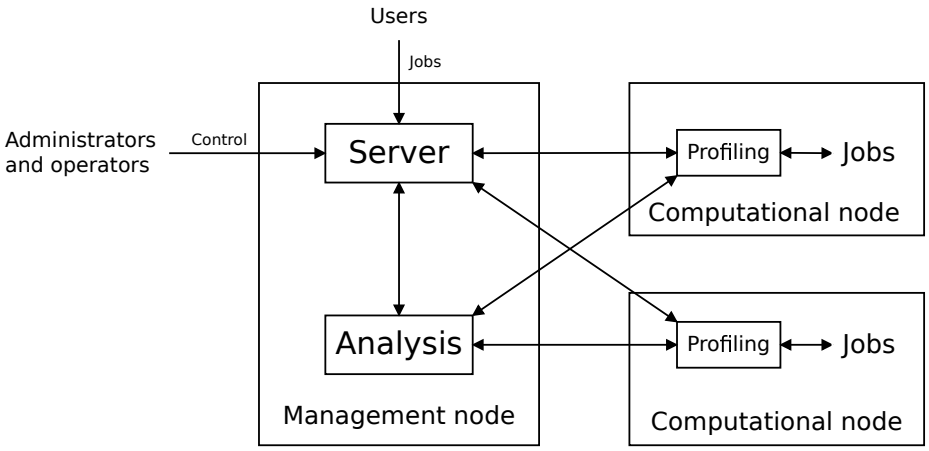

Figure 22. Profiling scheduler work flow.

	PBS	Profiling scheduler
Well-balanced	100%	102%
Overload	100%	85%

Table 9. Time spent running well-balanced and non-optimal jobs (with overused resources) with PBS and profiling system.

Profiling applications involves only a small overhead which is lesser or equal to 2% of the total wall clock time of a job (Table 9). Jobs that repeatedly start many short-living processes give more overhead, however, these types of jobs are rare. Profiling scheduler efficiently handles non-optimal jobs without losing overall resource utilization (Table 9).

Although, the system is developed mainly for Linux operating system there are possibilities to develop analogous systems for other UNIX platforms. All tests in this section were performed on a hybrid cluster (Table 8).

Conclusion

It is known that virtualization improves security, resilience to failures and eases administration substantially due to dynamic load balancing [9] without introducing substantial overheads. Moreover, a proper choice of virtualization package can improve CPU utilization.

The use of standard cloud technologies combined with process migration techniques can improve overall throughput of a distributed system and adapt it to problems being solved. In that way a virtual supercomputer can help people efficiently run applications and focus on domain-specific problems rather than on underlying computer architecture and placement of parallel tasks. Moreover, described approach can be beneficial in utilizing stream processors and GPU accelerators dynamically assigning them to virtual machines.

Virtual workspace hides intricacies of distributed computing behind a virtual machine to

streamline and boost scientific research workflow. It provides a convenient way of accessing hardware and software resources using unified tools, consolidates experiment's data and offers options to dynamically extend available resources using either private virtual cluster or high performance dedicated machines and public university cluster. Resources are accessed universally and in a unified way.

The key idea of a virtual supercomputer is to harness all available HPC resources and to provide a user with convenient access to them. Such a challenge can be effectively faced only using contemporary virtualization technologies. They can materialize the long-term dream of having virtual supercomputer at your desk.

Event-driven approach biggest advantage is the ease of parallel and distributed programming. First of all, what is needed from a programmer is to develop a class to describe each independent task, create objects of that class and submit them to a queue. Programming in such a way does not involve thread and lock management and the system is flexible enough to have even the tiniest tasks executed in parallel. Second, relieving programmer from thread management makes it easy to debug this system. Each thread maintains its own log and any of both system and user events can be written to it so the sequence of events can be restored after the execution ends. Finally, with event-driven approach it is easy to write load distribution algorithm for your specific problem (or use an existing one). The only thing which is not done automatically is decomposition and composition of tasks, however, this problem requires higher layer of abstraction to solve.

Proposed load balancing approach increases efficiency when using heterogeneous devices: it adds nested level of parallelism by pipelining execution of problem parts on different devices. In other words, it separates different kinds of workload and processes them concurrently. This feature lets it outperform OpenMP and Hadoop on computation- and data-intensive problems.

Finally, scheduling legacy applications can be done using similar approach but with somewhat coarser granularity. Each job submitted to portable batch system is profiled and its optimal resource reservation is determined. For subsequent submissions, this reservation is used by default. In contrast to event-driven approach this one does not use pipelining but rather uses traditional parallel programming libraries and does not require rewriting or even recompiling application source code.

Acknowledgments

The research presented in the chapter was carried out using computational resources of Resource Center Computer Center of Saint-Petersburg State University within frameworks of grants of Russian Foundation for Basic Research (project No. 13-07-00747) and Saint Petersburg State University (projects No. 9.38.674.2013, 0.37.155.2014).

References

[1] AWS-Compatible Private Cloud. Available online: `https://www.eucalyptus.com/aws-compatibility`. Retrieved: 2015-01-21.

[2] Extend private cloud into Amazon Web Services with OpenNebula. Available online: https://cloudbestpractices.wordpress.com/2011/11/07/extend-private-cloud-into-amazon-web-services-with-opennebula-2/. Retrieved: 2015-01-21.

[3] Foreign data wrappers — PostgreSQL wiki. https://wiki.postgresql.org/wiki/Foreign_data_wrappers. Retrieved: 2015-01-21.

[4] Degtyarev A. and Gankevich I. Wave surface generation using OpenCL, OpenMP and MPI. In *Proceedings of 8th International Conference «Computer Science & Information Technologies»*, pages 248–251, 2011.

[5] Degtyarev A.B. and Reed A.M. Modelling of incident waves near the ship's hull (application of autoregressive approach in problems of simulation of rough seas). In *Proceedings of the 12th International Ship Stability Workshop*, 2011.

[6] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. STAPL: An adaptive, generic parallel C++ library. In *Languages and Compilers for Parallel Computing*, pages 193–208. Springer, 2003.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[8] Alexander Bogdanov and Mikhail Dmitriev. Creation of hybrid clouds. In *Proc. of 8th International Conference «Computer Science & Information Technologies»*, pages 235–237, 2011.

[9] A.V. Bogdanov, A.B. Degtyarev, I.G. Gankevich, V.Yu. Gayduchok, and V. I. Zolotarev. Virtual workspace as a basis of supercomputer center. In *Proceedings of 5th International Conference on Distributed Computing and Grid-Technologies in Science and Education*, pages 60–66, 2012.

[10] Jeffrey S Chase, David E Irwin, Laura E Grit, Justin D Moore, and Sara E Sprenkle. Dynamic virtual clusters in a grid site manager. In *Proc. of the 12th International Symposium on High Performance Distributed Computing*, pages 90–100. IEEE, 2003.

[11] Yang Chen, Tianyu Wo, and Jianxin Li. An efficient resource management system for on-line virtual cluster provision. In *Proc. of International Conference on Cloud Computing (CLOUD)*, pages 72–79. IEEE, 2009.

[12] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[13] A. Degtyarev and I. Gankevich. Evaluation of hydrodynamic pressures for autoregression model of irregular waves. In *Proceedings of 11th International Conference on Stability of Ships and Ocean Vehicles, Athens*, pages 841–852, 2012.

[14] Alexander B Degtyarev and Arthur M Reed. Synoptic and short-term modeling of ocean waves. *International Shipbuilding Progress*, 60(1):523–553, 2013.

[15] Wesley Emeneker and Dan Stanzione. Dynamic virtual clustering. In *Proc. of International Conference on Cluster Computing*, pages 84–90. IEEE, 2007.

[16] Renato J Figueiredo, Peter A Dinda, and José AB Fortes. A case for grid computing on virtual machines. In *Proc. of the 23rd International Conference on Distributed Computing Systems*, pages 550–559. IEEE, 2003.

[17] Kazushige Goto and Robert Van De Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[18] Kazushige Goto and Robert Van De Geijn. High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4, 2008.

[19] Kevin Hamlen, Murat Kantarcioglu, Latifur Khan, and Bhavani Thuraisingham. Security issues for cloud computing. *International Journal of Information Security and Privacy (IJISP)*, 4(2):36–48, 2010.

[20] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[21] T Yu James. Performance evaluation of Linux Bridge. In *Telecommunications System Management Conference*, 2004.

[22] Vladimir V Korkhov, Jakub T Moscicki, and Valeria V Krzhizhanovskaya. The user-level scheduling of divisible load parallel applications with resource selection and adaptive workload balancing on the Grid. *Systems Journal*, 3(1):121–130, 2009.

[23] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[24] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose AB Fortes, and Renato J Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Proc. of the ACM/IEEE Supercomputing Conference*, page 7. IEEE, 2004.

[25] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[26] Andréa M Matsunaga, Maurício O Tsugawa, Sumalatha Adabala, Renato J Figueiredo, Herman Lam, and José AB Fortes. Science gateways made easy: the In-VIGO approach. *Concurrency and Computation: Practice and Experience*, 19(6):905–919, 2007.

[27] Hideo Nishimura, Naoya Maruyama, and Satoshi Matsuoka. Virtual clusters on the fly-fast, scalable, and flexible installation. In *Proc. of the 7th International Symposium on Cluster Computing and the Grid (CGRID)*, pages 549–556. IEEE, 2007.

[28] Manuel Rodríguez, Daniel Tapiador, Javier Fontán, Eduardo Huedo, Rubén S Montero, and Ignacio M Llorente. Dynamic provisioning of virtual clusters for grid computing. In *Euro-Par 2008 Workshops-Parallel Processing*, pages 23–32. Springer, 2009.

[29] Peter Sempolinski and Douglas Thain. A comparison and critique of Eucalyptus, OpenNebula and Nimbus. In *Proc. of the 2nd International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 417–426. IEEE, 2010.

[30] Larry Smarr and Charles E Catlett. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[31] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: The condor experience. *Concurrency and Computation: Practice and Experience*, 17(2-4):323–356, 2005.

[32] Peter Troger, Hrabri Rajic, Andreas Haas, and Piotr Domagalski. Standardization of an API for distributed resource management systems. In *Proc. of the 7th International Symposium on on Cluster Computing and the Grid (CCGRID)*, pages 619–626. IEEE, 2007.

[33] Steve Vinoski. Advanced message queuing protocol. *Internet Computing*, 10(6):87–89, 2006.

[34] Kejiang Ye, Xiaohong Jiang, Siding Chen, Dawei Huang, and Bei Wang. Analyzing and modeling the performance in Xen-based virtual cluster environment. In *Proc. of the 12th International Conference on High Performance Computing and Communications (HPCC)*, pages 273–280. IEEE, 2010.

[35] Dmitry Zotkin and Peter J Keleher. Job-length estimation and performance in backfilling schedulers. In *Proc. of the 8th International Symposium on High Performance Distributed Computing*, pages 236–243. IEEE, 1999.